The LEDA Data Type
Sorted Sequences
(sortseq)

Kurt Mehlhorn and Stefan Näher
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany

Contents

1 Abstract

We define and implement the LEDA data type sorted sequence (*sortseq*). An instance S of type sortseq is a sequence of items (of type *seq_item*). Every item contains a key from a linearly ordered data type K, called the key type of S, and an information from a data type I, called the information type of S. For every k in K there is at most one item in S with key k and the keys in a sorted sequence are increasing from front to rear of the sequence.

Sorted sequences are a very powerful data type. They can do everything that dictionaries and priority queues can do. In addition, they support *finger* searches and operations *conc*, *split*, *merge* and *delete_subsequence*.

Sorted sequences are implemented by skip lists. The dictionary and priority queue operations run in logarithmic time. Many of the additional operations, e.g., finger searches, *conc* and *split*, run in sublogarithmic time. For example, to concatenate two sequences of length n_1 and n_2 respectively takes time $O(\log \min(n_1, n_2))$.

The structure of this document as follows. We start with the exact definition of the data type sortseq by its LEDA manual page[N9̈5]. Then we give the h-files of the abstract data type sortseq and the concrete data type fskiplist and implement the abstract type in terms of the concrete data structure. The remainder of the document defines skip lists and gives the implementations of all operations on skip lists.

2 The Manual Page of Type Sorted Sequence

1. Definition

An instance S of the parameterized data type *sortseq*<K, I> is a sequence of items (*seq_item*). Every item contains a key from a linearly ordered data type K, called the key type of S, and an information from a data type I, called the information type of S. The number of items in S is called the size of S. A sorted sequence of size zero is called empty. We use $\langle k, i\rangle$ to denote a *seq_item* with key k and information i (called the information associated with key k). For each k in K there is at most one item $\langle k, i\rangle$ in S and if item $\langle k1, i1\rangle$ precedes item $\langle k2, i2\rangle$ in S then $k1 < k2$.

Sorted sequences are a very powerful data type. They can do everything that dictionaries and priority queues can do. In addition, they support finger searches and operations *conc*, *split*, *merge* and *delete_subsequence*.

The key type K must be linearly ordered. The linear order on K may change over time subject to the condition that the order of the elements that are currently in the sorted sequence remains stable. More precisely, whenever an operation (except for *reverse_items*) is applied to a sorted sequence S, the keys of S must form an increasing sequence according to the currently valid linear order on K. For operation *reverse_items* this must hold after the execution of the operation. An application of sorted sequences where the linear order on the keys evolves over time is the plane sweep algorithm for line segment intersection. This algorithm sweeps an arrangement of segments by a vertical sweep line and keeps the intersected segments in a sorted sequence sorted according to the y-coordinates of their intersections with the sweep line. For intersecting segments this order depends on the position of the sweep line.

Sorted sequences support finger searches. A finger search takes an item *it* in a sorted sequence and a key k and searches for the key starting at the item. The cost of a finger search is proportional to the logarithm of the distance of the key from the start of the search. A finger search is performed (!!) in the sequence containing the item and not in the main argument of a call *S.finger_search*(*it*, k). The sole role of S is to provide the type of the sorted sequence; S may or may not be the sequence containing *it*. It must however have the same type as the sequence containing *it*.

2. Creation

sortseq<K, I> S( );	creates an instance S of type *sortseq*<K, I> and initializes it to the empty sorted sequence.

3. Operations

K	S.key(*seq_item it*)	returns the key of item *it*. *Precondition*: *it* is an item in S.
I	S.inf(*seq_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in S.
seq_item	S.lookup(K k)	returns the item with key k (*nil* if no such item exists in S)
seq_item	S.finger_lookup(K k)	equivalent to *S.lookup*(k)
seq_item	S.locate(K k)	returns the item $\langle k1, i\rangle$ in S such that $k1$ is minimal with $k1 \geq k$ (*nil* if no such item exists).

seq_item S.finger_locate(K k) — equivalent to $S.locate(k)$

seq_item S.locate_succ(K k) — equivalent to $S.locate(k)$

seq_item S.succ(K k) — equivalent to $S.locate(k)$

seq_item S.finger_locate_succ(K k)
equivalent to $S.locate(k)$

seq_item S.locate_pred(K k) — returns the item $\langle k1, i\rangle$ in S such that $k1$ is maximal with $k1 \leq k$ (*nil* if no such item exists).

seq_item S.pred(K k) — equivalent to $S.locate\_pred(k)$

seq_item S.finger_locate_pred(K k)
equivalent to $S.locate\_pred(k)$

seq_item S.finger_lookup(*seq_item* it, K k)
equivalent to $IT.lookup(k)$ where IT is the sorted sequence containing it.
Precondition: S and IT must have the same type

seq_item S.finger_locate(*seq_item* it, K k)
equivalent to $IT.locate(k)$ where IT is the sorted sequence containing it.
Precondition: S and IT must have the same type.

seq_item S.finger_locate_succ(*seq_item* it, K k)
equivalent to $IT.locate\_succ(k)$ where IT is the sorted sequence containing it.
Precondition: S and IT must have the same type

seq_item S.finger_locate_pred(*seq_item* it, K k)
equivalent to $IT.locate\_pred(k)$ where IT is the sorted sequence containing it.
Precondition: S and IT must have the same type.

seq_item S.min_item( ) — returns the item with minimal key (*nil* if S is empty).

seq_item S.max_item( ) — returns the item with maximal key (*nil* if S is empty).

seq_item S.succ(*seq_item* it) — returns the successor item of it in the sequence containing it (*nil* if there is no such item).

seq_item S.pred(*seq_item* x) — returns the predecessor item of it in the sequence containing it (*nil* if there is no such item).

seq_item S.insert(K k, I i) — associates information i with key k: If there is an item $\langle k, j\rangle$ in S then j is replaced by i, else a new item $\langle k, i\rangle$ is added to S. In both cases the item is returned.

seq_item S.insert_at(*seq_item it*, *K k*, *I i*)

Like $IT.insert(k, i)$ where IT is the sequence containing item it.
Precondition: it is an item in IT with either
$key(it)$ is maximal with $key(it) < k$ or
$key(it)$ is minimal with $key(it) > k$.
S and IT must have the same type.

seq_item S.insert_at(*seq_item it*, *K k*, *I i*, *int dir*)

Like $IT.insert(k, i)$ where IT is the sequence containing item it.
Precondition: it is an item in IT with
$key(it)$ is maximal with $key(it) < k$ if $dir\ =\ before$ or
$key(it)$ is minimal with $key(it) > k$ if $dir\ =\ after$.
S and IT must have the same type.

int S.size( )   returns the size of S.

bool S.empty( )   returns true if S is empty, false otherwise.

void S.clear( )   makes S the empty sorted sequence.

void S.reverse_items(*seq_item a*, *seq_item b*)

the subsequence of IT from a to b is reversed, where IT is the sequence containing a and b.
Precondition: a appears before b in IT.

void S.flip_items(*seq_item a*, *seq_item b*)

equivalent to $S.reverse\_items(a, b)$.

void S.del(*K k*)   removes the item with key k from S (null operation if no such item exists).

void S.del_item(*seq_item it*)   removes the item it from the sequence containing it.

void S.change_inf(*seq_item it*, *I i*)

makes i the information of item it.

void S.split(*seq_item it*, *sortseq<K, I>& S1*, *sortseq<K, I>& S2*, *int dir*)

splits IT at item it, where IT is the sequence containing it, into sequences *S1* and *S2* and makes IT empty (if distinct from *S1* and *S2*). More precisely, if $IT = x_1, \ldots, x_{k-1}, it, x_{k+1}, \ldots, x_n$ and $dir\ =\ after$ then $S1\ =\ x_1, \ldots, x_{k-1}, it$ and $S2\ =\ x_{k+1}, \ldots, x_n$. If $dir\ =\ before$ then *S2* starts with it after the split.

void S.delete_subsequence(*seq_item a*, *seq_item b*, *sortseq<K, I>& S1*)

deletes the subsequence starting at a and ending at b from the sequence IT containing both and assigns the subsequence to *S1*.
Precondition: a and b belong to the same sequence IT, a is equal or before b and IT and *S1* have the same type.

sortseq<K, I>& S.conc(*sortseq*<K, I>& *S1*, *int dir*)

appends *S1* at the front (dir = *before*) or rear (dir = *after*) end of S, makes *S1* empty and returns S.
Precondition: $S.key(S.max\_item(\,)) < S1.key(S1.min\_item(\,))$ if dir = *after* and $S1.key(S1.max\_item(\,)) < S.key(S.min\_item(\,)$ if dir = *before*.

void S.merge(*sortseq*<K, I>& *S1*)

merges the sequence *S1* into sequence S and makes *S1* empty.
Precondition: all keys are distinct.

void S.print(*ostream*& *out*, *string s*, *char c* = ' ')

prints s and all elements of S separated by c onto stream *out*.

void S.print(*string s*, *char c* = ' ')

equivalent to $S.print(cout, s, c)$.

bool $S \equiv S1$ returns *true* if S agrees with *S1* componentwise and *false* otherwise

4. Implementation

Sorted sequences are implemented by skiplists [Pug89, Pug90]. Let n denote the current size of the sequence. Operations *insert*, *locate*, *lookup* and *del* take time $O(\log n)$, operations *succ*, *pred*, *max*, *min_item*, *key*, *inf*, *insert_at_item* and *del_item* take time $O(1)$. *Clear* takes time $O(n)$ and *reverse_items* takes time $O(l)$, where l is the length of the reversed subsequence. *Finger_lookup*(x) and *finger_locate*(x) take time $O(\log \min(d, n-d))$ if x is the d-th item in S. *Finger_lookup*(it, x) and *finger_locate*(it, x) take time $O(\log \min(d, n-d))$ where d is the number of items between *it* and the item containing x. Note that $\min(d, n-d)$ is the smaller of the distances from *it* to x if sequences are viewed as circularly closed. *Split*, *delete_subsequence* and *conc* take time $O(\log \min(n_1, n_2))$ where n_1 and n_2 are the sizes of the results of *split* and *delete_subsequence*, respectively the sizes of the arguments of *conc*. *Merge* takes time $O(\log((n_1 + n_2)/n_1))$ where n_1 and n_2 are the sizes of the two arguments. The space requirement of sorted sequences is linear in the length of the sequence (about $40n$ Bytes for a sequence of size n plus the space for the keys and the informations.).

5. Example

We use a sorted sequence to list all elements in a sequence of strings lying lexicographically between two given search strings.

```
#include <LEDA/sortseq.h>
main()
{ sortseq<string,int> S;
  string s1,s2;
  while ( cin >> s1 &&  s1 != "stop" ) S.insert(s1,0);
  while ( cin >> s1 >> s2 )
    { seq_item stop = S.locate(s2);
      for (seq_item it = S.locate(s1); it != stop; it = S.succ(it))
        cout << S.key(it) << "\n";
    }
}
```

Further examples can be found in section Sorted Sequences of [MN94].

3 Header files

We define two header files, a header file sorteq.h for the abstract class *sortseq*<K, I> and a header file fskiplist.h for the concrete class *fskiplist*. In the concrete class the keys and infs are generic pointers.

3.1 Sortseq.h

The purpose of sortseq.h is to define the abstract class *sortseq*<K, I> and to implement its functions in terms of the functions of the implementation class *fskiplist*. More precisely, every abstract function (e.g. *lookup*) calls the concrete function with the same name after converting any arguments of type K or I to a generic pointer (by means of function *leda_cast*) and after converting any argument of type *sortseq*<K, I> to a *fskiplist* (by a cast). Similarly, any result of type K or I is converted back from generic pointer (by means of the `LEDA_ACCESS` macro).

In the definition of class *sortseq*<K, I> we also introduce the functions *cmp*, *clear_key*, *clear_inf*, *copy_inf*, *print_key*, *print_inf* and *int_type*. They take generic pointers, convert them to types K and I respectively and then perform the action indicated by their name; we refer the reader to [MN94, chapter Implementation] for more details. The functions above are declared virtual in class *fskiplist*. When *sortseq* is derived from *fskiplist* the virtual definitions are overwritten by the definitions in class *sortseq*. In order to double-check we have included error messages in the bodies of the virtual functions. We come back to this point in sections 3.1, 4.3 and 8. weglassen?

⟨*sortseq.h*⟩≡

```
#ifndef F_SORTSEQ_H
#define F_SORTSEQ_H

#include <LEDA/basic.h>
#include "fskiplist.h"
#define IMPL fskiplist
typedef skiplist_item seq_item;

template<class K, class I>

class sortseq : public virtual IMPL {

int  int_type()              const  {  return LEDA_INT_TYPE(K);  }
int  cmp(GenPtr x, GenPtr y) const  {  return LEDA_COMPARE(K,x,y);  }
void clear_key(GenPtr& x)    const  {  LEDA_CLEAR(K,x);  }
void clear_inf(GenPtr& x)    const  {  LEDA_CLEAR(I,x);  }
void copy_key(GenPtr& x)     const  {  LEDA_COPY(K,x);  }
void copy_inf(GenPtr& x)     const  {  LEDA_COPY(I,x);  }
void print_key(GenPtr x)     const  {  LEDA_PRINT(K,x,cout);  }
void print_inf(GenPtr x)     const  {  LEDA_PRINT(I,x,cout);  }

public:

sortseq(const sortseq<K,I>& S1) : IMPL(S1){ }

sortseq(){ }


virtual K  key(seq_item it)  const  { return LEDA_ACCESS(K,IMPL::key(it)); }

virtual I  inf(seq_item it)  const  { return LEDA_ACCESS(I,IMPL::inf(it)); }

virtual seq_item lookup(K k) const  {  return IMPL::lookup(leda_cast(k));  }
```

```
virtual seq_item finger_lookup(K k) const
        {   return IMPL::finger_lookup(leda_cast(k));   }

virtual seq_item locate(K k) const  {  return IMPL::locate(leda_cast(k));  }

virtual seq_item finger_locate(K k) const
        {   return IMPL::finger_locate_succ(leda_cast(k));   }

virtual seq_item locate_succ(K k) const
        {  return IMPL::locate_succ(leda_cast(k));  }

virtual seq_item succ(K k) const  {  return locate_succ(k);  }

virtual seq_item finger_locate_succ(K k) const
        {   return IMPL::finger_locate_succ(leda_cast(k));   }

virtual seq_item locate_pred(K k) const
        {  return IMPL::locate_pred(leda_cast(k));  }

virtual seq_item pred(K k) const  {  return locate_pred(k);  }

virtual seq_item finger_locate_pred(K k) const
        {   return IMPL::finger_locate_pred(leda_cast(k));   }

virtual seq_item finger_lookup(seq_item it,K k) const
        {   return IMPL::finger_lookup(it,leda_cast(k));   }

virtual seq_item finger_locate(seq_item it,K k) const
        {   return IMPL::finger_locate_succ(it,leda_cast(k));   }

virtual seq_item finger_locate_succ(seq_item it,K k) const
        {   return IMPL::finger_locate_succ(it,leda_cast(k));   }

virtual seq_item finger_locate_pred(seq_item it,K k) const
        {   return IMPL::finger_locate_pred(it,leda_cast(k));   }

virtual seq_item min_item() const  {  return IMPL::min_item();  }

virtual seq_item max_item() const  {  return IMPL::max_item();  }

virtual seq_item succ(seq_item it) const  {  return IMPL::succ(it);  }

virtual seq_item pred(seq_item x) const  {  return IMPL::pred(x);  }

virtual seq_item insert(K k,I i)
        {  return IMPL::insert(leda_cast(k),leda_cast(i));  }

virtual seq_item insert_at(seq_item it,K k,I i)
        {  return IMPL::insert_at_item(it,leda_cast(k),leda_cast(i));  }

virtual seq_item insert_at(seq_item it,K k,I i,int dir)
        {  return IMPL::insert_at_item(it,leda_cast(k),leda_cast(i),dir);  }
```

```
virtual int      size()    {  return IMPL::size();  }

virtual bool     empty() const {  return IMPL::empty(); }

virtual void     clear()  {  IMPL::clear();  }

virtual void reverse_items(seq_item a, seq_item b)
        {  IMPL::reverse_items(a,b);  }

virtual void flip_items(seq_item a, seq_item b)  {  reverse_items(a,b);  }

virtual void del(K k)   {  IMPL::del(leda_cast(k));  }

virtual void del_item(seq_item it)   {  IMPL::del_item(it);  }

virtual void change_inf(seq_item it, I i)  {  IMPL::change_inf(it,leda_cast(i)); }

virtual void split(seq_item it,sortseq<K,I>& S1,sortseq<K,I>& S2,int dir)
        {  IMPL::split_at_item(it,(IMPL&)S1,(IMPL&)S2,dir);  }

virtual void delete_subsequence(seq_item a,seq_item b,sortseq<K,I>& S1)
        {  IMPL::delete_subsequence(a,b,(IMPL&)S1);  }

virtual sortseq<K,I>& conc(sortseq<K,I>& S1,int dir)
        {  IMPL::conc((IMPL&)S1,dir); return *this;  }

virtual void merge(sortseq<K,I>& S1) { IMPL::merge((IMPL&)S1);  }

virtual void print(ostream& out,string s,char c=' ') const
        {  IMPL::print(out,s,c);  }

virtual void print(string s,char c=' ') const
        {  IMPL::print(cout,s,c);  }

virtual void check_data_structure(string s)
        {  IMPL::check_data_structure(s);  }
sortseq<K,I>& operator=(const sortseq<K,I>& S1)
        {  IMPL::operator=((IMPL &) S1); return *this;  }
/*
assigns a copy of |S1| to |S|.
*/
bool operator==(const sortseq<K,I>& S1)
{ seq_item sit = min_item();  seq_item sit1 = S1.min_item();
  while (sit != nil && sit1 != nil)
   if (compare(key(sit),S1.key(sit)) == 0)
     { sit = succ(sit);  sit1 = S1.succ(sit1);}
   else return false;
  return (sit == nil && sit1 == nil);
}

bool operator!=(const sortseq<K,I>& S1)  {  return (!(operator==(S1)));  }
/*
```

```
returns |false| if |S| agrees with |S1| componentwise and |true| otherwise
*/
virtual ~sortseq()    {  IMPL::clear();  }
}  ;
#endif
```

There are two additional points in sortseq.h to which we want to allude the reader: the implementation of the destructor and the function *check_data_structure*. The latter function checks whether the current concrete data structure satisfies all the invariants of a skiplist. We use it in our test programs but do not list it in the manual. The destructor $\sim$*sortseq* deserves close attention. It first calls *fskiplist*:: *clear* and then the destructor $\sim$*fskiplist*. The latter call is generated by the system since *sortseq* is derived from *fskiplist*. It does not suffice to call $\sim$*fskiplist* since the constructors and destructors of the base class use the original definition of the virtual functions *clear_key* and *clear_inf* given in fskiplist.h and not the definition given in the derived class. As a consequence the destructor $\sim$*fskiplist* does not know how to destroy a key or inf and therefore should only be called for an empty skiplist. The same effect plays a role in the implementation of the copy constructor for fskiplists; cf. section 4.3.

3.2 Fskiplist.h

File fskiplist.h defines class *fskiplist*. The public part of this class definition is in one-to-one correspondance to the public part of class *sortseq*. In addition we define the virtual functions *copy_key*, *copy_inf*, *clear_key*, *clear_inf*, *cmp*, and *int_type* and some private functions which are used in the implementation of the public functions and are explained as we go along. We also need to declare the types *skiplist_item* and *large_item* which we need to define the data members of class fskiplist. The virtual functions are overwritten in the definition of class *sortseq*$<K, I>$ and hence most of them will never be called. Exeptions are *copy_inf* and *copy_key* which are called in the copy-constructor, cf. section 4.3.

⟨*fskiplist.h*⟩≡

```
#ifndef F_SKIPLIST_H
#define F_SKIPLIST_H
#include <LEDA/basic.h>
class header_node;
class fskiplist_node;
class fkiplist;
typedef fskiplist_node* skiplist_item;
typedef header_node* large_item;
⟨definition of classes fskiplist node and header node⟩;
class fskiplist
{
⟨data members of class fskiplist⟩
int  randomLevel();
virtual int cmp(GenPtr x, GenPtr y) const
        { error_handler(1,"cmp should never be called"); return 0; }
virtual void copy_key(GenPtr&)  const  { }
virtual void copy_inf(GenPtr&)  const  { }
virtual void clear_key(GenPtr&) const
        { error_handler(1,"clear_key should never be called");  }
virtual void clear_inf(GenPtr&) const
        { error_handler(1,"clear_inf should never be called");  }
```

```
virtual void print_key(GenPtr)  const
        { error_handler(1,"print_key should never be called");  }
virtual void print_inf(GenPtr)  const
        { error_handler(1,"print_inf should never be called");  }

virtual int int_type()    const  {
   error_handler(1,"int_type should never be called"); return 0;  }

skiplist_item search(skiplist_item,int,GenPtr,int&) const;
skiplist_item finger_search(skiplist_item,GenPtr,int&) const;
skiplist_item finger_search(GenPtr,int&) const;
void          remove_item(skiplist_item);
void          insert_item_at_item(skiplist_item,skiplist_item,int);
void          print(const fskiplist &, ostream&, string s, char space) const;
void          check_data_structure(const fskiplist&, string s);


public:
 fskiplist(float prob = 0.25);
 fskiplist(const fskiplist&);
 fskiplist& operator=(const fskiplist&);
 virtual ~fskiplist();

 GenPtr key(skiplist_item p) const;
 GenPtr inf(skiplist_item p) const;
 GenPtr& info(skiplist_item p) const;
 int     get_height(skiplist_item p) const;

 skiplist_item insert(GenPtr key, GenPtr inf);

 skiplist_item locate(GenPtr key) const;
 skiplist_item locate_succ(GenPtr key) const;
 skiplist_item locate_pred(GenPtr key) const;
 skiplist_item lookup(GenPtr key) const;

 skiplist_item finger_locate(skiplist_item, GenPtr key) const;
 skiplist_item finger_locate_succ(skiplist_item,GenPtr key) const;
 skiplist_item finger_locate_pred(skiplist_item,GenPtr key) const;
 skiplist_item finger_lookup(skiplist_item,GenPtr key) const;

 skiplist_item finger_locate(GenPtr key) const;
 skiplist_item finger_locate_succ(GenPtr key) const;
 skiplist_item finger_locate_pred(GenPtr key) const;
 skiplist_item finger_lookup(GenPtr key) const;

 skiplist_item min_item() const;
 skiplist_item max_item() const;
 void          reverse_items(skiplist_item,skiplist_item);
 void          del(GenPtr key);
 void          del1(GenPtr key);

 void          conc(fskiplist&,int);
 void          split_at_item(skiplist_item,fskiplist&,fskiplist&,int);

 void          merge(fskiplist&);

 void          delete_subsequence(skiplist_item,skiplist_item, fskiplist&);

 void          print(ostream& out, string s, char space) const
                 { print(*this,out,s,space); }

 void          check_data_structure(string s)
```

```
                  { check_data_structure(*this,s); }
 skiplist_item insert_at_item(skiplist_item p, GenPtr key, GenPtr inf);
 skiplist_item insert_at_item(skiplist_item p, GenPtr key, GenPtr inf, int dir);

 void          change_inf(skiplist_item p, GenPtr inf);
 void          del_item(skiplist_item p);
 void          clear();
 int           size() ;
 int           empty() const;

 skiplist_item first_item() const;
 skiplist_item next_item(skiplist_item p) const;
 skiplist_item succ(skiplist_item p) const;
 skiplist_item pred(skiplist_item p) const;

 /* piority queue operations */

 skiplist_item find_min() const;
 void del_min();
 void decrease_key(skiplist_item p, GenPtr k);

};

⟨implementation of inline functions⟩;
#endif
```

4 Skip Lists

4.1 Definition

We represent sorted sequences by skip lists [Pug89, Pug90]. In a skip list every element is represented by a tower (type *fskiplist_node*). A tower has a non-negative *height* and contains a *key*, an *inf*ormation, $h+1$ *forward* pointers to towers, a *backward* pointer and a *pred*ecessor pointer. Besides the towers representing the elements there are also two extra towers called *header* and *STOP*. We refer to them as improper towers. The height of *header* is *MaxHeight* and the height of *STOP* is -1. The *header* has type *header_node*. A header_node is a fskiplist_node with two additional fields: the field *true_height* is equal to one plus the maximal height of any tower in the list and the field *myseq* points to the skiplist to which *header* belongs.

If the sequence is empty then *true_height* is 0. We require that $true\_height \leq Maxheight$. Logically, the keys of the two improper towers are $-\infty$ and ∞ respectively, but note that these keys are not stored (it would make life easier but we cannot expect any linearly ordered type to actually have these elements).

The values of the various pointers are as follows. Let v be any tower and let h be the height of v (view *header* as a tower of height *true_height* for this paragraph). Then for all i, $0 \leq i \leq h$, the i-th forward pointer of v points to closest successor tower of height i or more (to *STOP* if there is no such tower), the backward pointer points to the node w with the highest forward pointer into v, i.e., the h-th forward pointer of w points to v, and the predecessor pointer of v points to the tower immediately preceeding v. The function *check_data_structure* checks whether these invariants are satisfied.

The height of a tower is chosen probabilistically when a tower is created. It is set to h with probability $p^h(1-p)$ where p with $0 < p < 1$ is a real parameter fixed when the skip list is created. In our implementation we use $1/4$ as the default value for p. In a skiplist with n items the expected maximal height of a tower is $\log_{1/p} n + O(1)$ since the probability that some tower has height k or more is bounded by $\max(1, np^k)$ for every integer k. Also, if v is any tower then the probability that $v \to backward$ has height larger than v is p. Thus after traversing an expected number of $1/p$ backward pointers starting in any node v a tower higher than v is reached.

⟨*definition of classes fskiplist node and ...*⟩≡

```
class fskiplist_node
{ friend class fskiplist;

  GenPtr key;
  GenPtr inf;
  int    height;
  skiplist_item  pred;
  skiplist_item  backward;
  skiplist_item* forward;  // pointer to array of forward pointers
};

class header_node : public fskiplist_node
{ friend class fskiplist;

  int true_height;
  fskiplist* myseq;

}
```

A *skiplist_item* is a pointer to a *fskiplist_node* and a *large_item* is a pointer to a *header_node*. Let v be any item. We have $v \to height \equiv MaxHeight$ if v is a header and $v \to height < 0$ if v is a stop node. If v is a header we can cast it to a large item (by $(large\_item)v$) and access the skiplist containing the header (by $((large\_item)v) \to myseq$).

In a fskiplist we store the items *header* and *STOP* and some quantities related to the random process: *prob* contains the parameter p in use, and *randomBits* contains a integer whose last *randomsLeft* bits are random.

⟨*data members of class fskiplist*⟩≡

```
    large_item  header;         // pointer to header
    skiplist_item STOP;         // pointer to end
    int randomBits;             // random bit source
    int randomsLeft;            // number of unused bits in randomBits

    float prob;
```

4.2 Implementation: An Overview

The basic structure of the skip list implementation is simple. We first define some constants and then implement the operations on skip lists.

Most operations on skip lists do no check their pre- and postconditions. If `CHECK_INVARIANTS` is defined then many operations do. For some operations this increases the running time by more than a constant factor.

⟨*fskiplist.c*⟩≡

```
#include "fskiplist.h"
#include </KM/usr/mehlhorn/Programs/basic.h>
⟨constants⟩;
⟨constructors and related functions⟩;
⟨search functions⟩;
⟨insert and delete functions⟩;
⟨concatenate and related functions⟩;
⟨miscellaneous⟩;
```

It is convenient to define two constants. *BitsinRandom* is the number of random bits in a random non-negative *int* (the value of such an int lies between 0 and $2^{31}-1$ inclusive) and *MaxHeight* is one larger than the maximal height on any skiplist node.

⟨*constants*⟩≡

```
const int BitsInRandom      = 31;
const int MaxHeight         = 32;
```

4.3 Constructors, Destructor, Assignment and Clear

Class fskiplist has two constructors. The first constructor constructs an empty skip list and the second constructor copies its argument.

Let us look more closely at the first constructor. We allocate a tower of height *MaxHeight* for *header* and a tower of height -1 for *STOP*. The *true_height* of the *header* is 0 and hence only the level 0 forward pointer of *header* is initialized.

The copy constructor first constructs an empty skip list and then copies its argument L element by element. Since the constructor of class fskiplist uses the trivial versions of *copy_key* and *copy_inf* defined in the definition of class fskiplist, i.e., the calls of *copy_key* and *copy_inf* in *insert_at_item* have no effect, we have to use L's version of these functions to do the copying.

The default constructor takes constant time and the copy constructor takes linear expected time plus the time to copy n keys and informations.

⟨*constructors ...*⟩≡

```
fskiplist::fskiplist(float p)
{ prob = p;
  randomBits = rand_int(0,MAXINT-1);
  randomsLeft = BitsInRandom;

  header = new header_node;
  header -> forward = new skiplist_item[MaxHeight+1];
  header ->height = MaxHeight;
  header -> true_height = 0;
  header->myseq = this;
  STOP = new fskiplist_node;
  STOP -> height = -1;
  STOP->backward= (skiplist_item) header;
  STOP->pred= (skiplist_item) header;
  header->forward[0] = STOP;
}

fskiplist::fskiplist(const fskiplist& L)
{ prob = L.prob;
  randomBits = rand_int(0,MAXINT-1);
  randomsLeft = BitsInRandom;
  header = new header_node;
  header -> forward = new skiplist_item[MaxHeight+1];
  header ->height = MaxHeight;
  header -> true_height = 0;
  header -> myseq = this;
  STOP = new fskiplist_node;
  STOP -> height = -1;
  STOP->backward= (skiplist_item) header;
  STOP->pred= (skiplist_item) header;

  header->forward[0] = STOP;

  skiplist_item p = L.STOP->pred;
```

```
  while (p!= L.header)
  { insert_at_item(header,p->key,p->inf);
    L.copy_key(p->key);
    L.copy_inf(p->inf);
    p = p->pred;
  }
}
```

The assignment operator first clears the skip list on the left side of the assignment and then copies the argument. The *clear* function deletes all nodes of a skip list and the destructor deletes in addition the two extra nodes. All three functions take linear expected time plus the time to copy or clear n keys and informations.

$\langle$*constructors ...*$\rangle$+$\equiv$

```
fskiplist& fskiplist::operator=(const fskiplist& L)
{ clear();
  skiplist_item p = L.STOP->pred;
  while (p!= L.header)
  { insert_at_item(header,p->key,p->inf,after);
    p = p->pred;
   }
  return *this;
 }

void fskiplist::clear()
{ register skiplist_item p,q;
  p = header->forward[0];
  while(p!=STOP)
  { q = p->forward[0];
    clear_key(p->key);
    clear_inf(p->inf);
    delete p-> forward;
    delete p;
    p = q;
   }
 header -> true_height = 0;
 header->forward[0] = STOP;
 STOP->pred= (skiplist_item) header;
}

fskiplist::~fskiplist()
{ clear();
  delete header -> forward;
  delete header;
  delete STOP;
 }
```

5 Search Operations

5.1 The Basic Search Routine

Skip lists offer a wide variety of search operations. The basic function is *search*. It takes a *key*, an item v and a height h such that v has height at least h and $v \rightarrow key < key \leq v \rightarrow forward[h] \rightarrow key$

and returns an item q and an integer l such that $q \to pred \to key < key \leq q \to key$ and $l \geq 0$ iff $key \equiv q \to key$.

The principle underlying *search* is simple. It maintains items p and q and a height k, $k \geq -1$, such that p's height is at least $k+1$, q is the level $k+1$ successor of p and $p \to key < key \leq q \to key$. If k is -1 then q is returned. If $k \geq 0$ then we search through level k starting at $q = p \to forward[k]$ to determine the new p and q.

```
q = p->forward[k];
while (key > q -> key)   {  p = q; q = p -> forward[k]; }
```

The basic strategy can be slightly optimized as follows. Before making a comparison between keys we check whether the current q has height k (otherwise, it is equal to the old candidate for q and the comparison was already made).

The expected running time of *search* is $O(1+h)$ since $h+1$ levels are visited and since the expected time spent on each level is constant. The easiest way to see the latter fact is to traverse the search path backwards and to recall that after following a constant expected number of backward pointers a higher tower is reached.

$\langle$*search ...*$\rangle\equiv$

```
skiplist_item fskiplist::search(skiplist_item v, int h, GenPtr key, int& l) const
{ register skiplist_item p = v;
  register skiplist_item q = p -> forward[h];
  l = 0;

#ifdef CHECK_INVARIANTS
if ((p -> height < MaxHeight && cmp(key,p -> key) <= 0) ||
    (q -> height >= 0 && cmp(key,q - >key) > 0))
       error_handler(1,"search: precondition violated");
#endif

  if (q -> height >= 0 && cmp(key,q -> key) == 0)  return q;

  int k = h - 1;
  int c = -1;

    while (k >= 0)
    { /* p -> key < key < p -> forward[k+1] -> key and c == -1 */
      q = p -> forward[k];
      while (k == q -> height && (c = cmp(key,q -> key)) > 0)
      { p = q;
        q = p -> forward[k];
       }
      if(c == 0) return q;
      k--;
     }

#ifdef CHECK_INVARIANTS
p = q->pred;
if (( p ->height < MaxHeight && cmp(key,p->key) <= 0) ||
    ( q->height  >= 0 && cmp(key, q->key) > 0) ||
    ( k >= 0 && cmp(key,q->key) != 0)||
    ( k < 0 && q->height  >= 0 && cmp(key,q->key) == 0 ))
    error_handler(1, "seach: postcondition is violated");
```

```
#endif
  l = k;
  return q;
 }
```

It is easy to derive the other search functions from the basic routine *search*. The call *locate_succ*(k) returns the item $\langle k1, i\rangle$ with $k \le k1$ and $k1$ minimal (*nil* if there is no such item), *locate_pred* is symmetric to *locate_succ*, *locate* is synonymous to *locate_succ* and *lookup*(k) returns the item $\langle k, i\rangle$ (*nil* if there is no such item). All operations in this section take logarithmic time.

$\langle$*search ...*$\rangle{+}{\equiv}$

```
skiplist_item fskiplist::locate_succ(GenPtr key) const
{ int l;
  skiplist_item q = search(header,header->true_height,key,l);
  return (q==STOP) ? 0 : q;
 }
skiplist_item fskiplist::locate(GenPtr key) const
{ return locate_succ(key); }
skiplist_item fskiplist::locate_pred(GenPtr key) const
{ int l;
  skiplist_item q = search(header,header->true_height,key,l);
  if (l < 0) q = q->pred;
  return (q==header) ? 0 : q;
 }
skiplist_item fskiplist::lookup(GenPtr key) const
{ int k;
  skiplist_item q = search(header,header->true_height,key,k);
  return (k<0) ? 0 : q;
 }
```

5.2 Finger Searches

We turn to finger searches. We desribe two versions.

The first version takes a *key* and locates an item q and an integer l such that $q \to pred \to key < key \le q \to key$ and $l \ge 0$ iff $key \equiv q \to key$ and runs in time $O(\log \min(d, n-d))$ if q is the d-th item in a list of n items.

The idea is to perform a simultaneous (also called dove-tailed) search for *key* from the last and the first item.

The time bound can be seen as follows. If q is the d-th item then the expected maximal height of a tower preceeding q is $\log d$ and the expected maximal height of a tower succeeding q is $\log n - d$. Thus $\log \min(d, n-d)$ is the expected maximal value of the variable k in the program below. The time bound follows.

$\langle$*search ...*$\rangle{+}{\equiv}$

```
skiplist_item fskiplist::finger_search(GenPtr key, int& l) const
{ skiplist_item q = STOP->pred;
if ( q == header || cmp(key, q->key) >0)
   {  l = -1;
      return STOP; }
/* we have header->key  < key <= q->key */
header -> key = key;  STOP -> key = key;
l = 0;
```

```
int k = 0;
int c1,c2;

while (((c1 = cmp(key,header ->forward[k]->key)) > 0) &&
       ((c2 = cmp(key, q -> key)) <= 0) )
  {  /* q is the righmost tower of height at least k; it it proper
        since k == height implies header->forward[k] == STOP and the loop
        body would not have been entered. */

    k++;
    if (c2 == 0)  return q;
    while (k > q -> height)  q = q -> backward;
  }

if (c1 == 0 && STOP != header ->forward[k])
return header -> forward[k];

if (c1 <= 0)
  {
#ifdef CHECK_INVARIANTS
    skiplist_item r = header->forward[k];
    if (r!= STOP && cmp(key, r->key) >= 0)
                  error_handler(1,"dove-tailed: first case");
#endif
    return search(header,k,key,1);
  }
#ifdef CHECK_INVARIANTS
skiplist_item r = q->forward[k];
if (( q != header && cmp(key, q->key) <= 0) ||
    ( r != STOP && cmp(key, r->key) >=0 ) )
        error_handler(1,"dove-tailed: second case");
#endif
return search(q,k,key,1);
}
```

The second version of finger search takes an item v and a *key* and returns an integer l and an item q such that $q \to pred \to key < key \leq q \to key$ and $l \geq 0$ iff $key \equiv q \to key$. It runs in time $O(\log \min(d, n-d))$ where d is the number of items between v and q.

The strategy used by *finger_search* is simple. A standard search starts with an item p and an integer k such that $p \to key < key \leq p \to forward[k] \to key$. In a finger search we start with an arbitrary item p and hence have one of the two inequalities with $k \equiv 0$.

Assume first that we have the first inequality for some $k \geq 0$. If we also have the second we are done and can proceed as in a standard search. Otherwise we proceed to the predecessor of p of height $k+1$. There we still have the first inequality and are 'closer' to also having the second since the forward pointer now has a wider reach.

Assume next that we have the second inequality (with inequality) for some $k \geq 0$. If we also have the first or if $key \equiv p \to key$ then we are done. Otherwise we move to $p \to backward$. There we still have the second inequality and are 'closer' to having the first. Furthermore, after an expected constant number of steps we reach a higher level and therefore make larger strides.

We proceed as describes above until we either have both inequalities or reach one of the sentinels. In the former case we start a standard search and in the latter case we start the dove-tailed search of the preceeding section.

The running time of *finger_search* is readily determined. Assume for simplicity that q is right of v (the other case being symmetric) and that v is the n_1-th item in the sequence. Then v and q split the list into three parts of length n_1, $n_2 = d$, and $n_3 = n - n_1 - n_2$ respectively. Use h_i to denote the maximal height of a tower in the i-th part. Then $E[h_i] = \log n_i + O(1)$. The height reached by the backward walk is $h_0 = \min(h_1, h_2) = \log \min(d, n-d)$. If the backward walk reaches a node satisfying both ineqalities then the second part of the search is a standard search that takes time $O(h_0)$ as well. If the backward walk reaches the header then $h_1 \leq h_2$ and the second part of the search is the dove-tailed search of the preceeding section that takes time $\min(\max(h_1, h_2), h_3)) = \min(h_2, h_3) = \log \min(d, n-d)$.

$\langle$*search ...*$\rangle{+}\equiv$

```
skiplist_item fskiplist::finger_search(skiplist_item v, GenPtr key, int& l) const
{ l = 0;
  skiplist_item p =v;
  if (p->height  < 0 ) p = p->backward;
  if (p->height == MaxHeight)
        return ((large_item) p)->myseq->finger_search(key,l);
  int dir = cmp(key, v->key);
  if  (dir == 0) return v;
  int k = 0;
  int c ;
  if (dir > 0)
    {   while (p->height < MaxHeight && p->forward[k]->height  >= 0 &&
                (c = cmp(key,p -> forward[k] ->key )) >= 0)
        { if (c == 0) return p->forward[k];
          k++;
          while (k > p->height) p = p->backward;
        }
        if (p->height == MaxHeight)
            return  ((large_item) p)->myseq->finger_search(key,l);
     }
  else
    {  while (p->height  < MaxHeight && p->forward[k] ->height >= 0
                && (c = cmp(key, p->key)) <= 0)
        { if (c == 0)  return p;
          k = p -> height;
          p = p -> backward;
        }
       if (p->forward[k]->height  < 0 )
         { p = p->forward[k]->backward;
           return ((large_item) p)->myseq->finger_search(key,l);
         }
     }
#ifdef CHECK_INVARIANTS
if ((p->height  < MaxHeight && cmp(key, p->key) <= 0) ||
   ( p->forward[k]->height  >= 0 && cmp(key, p->forward[k] ->key) >= 0))
error_handler(1,"violated invariant in finger_search");
#endif
return search(p,k,key,l);
}
```

We can now derive functions *finger_locate_succ*, *finger_locate_pred* and *finger_lookup* as in the case of the standard search. All three function have running time $O(\log \min(d, n-d))$, where d is defined as in one of the two preceeding sections.

⟨*search ...*⟩+≡

```
skiplist_item fskiplist::finger_locate_succ(skiplist_item v, GenPtr key) const
{ int l;
  skiplist_item q = finger_search(v,key,l);

  return (q->height < 0) ? 0 : q;
 }

skiplist_item fskiplist::finger_locate_succ(GenPtr key) const
{ int l;
  skiplist_item q = finger_search(key,l);

  return (q==STOP) ? 0 : q;
 }

skiplist_item fskiplist::finger_locate_pred(skiplist_item v, GenPtr key) const
{ int l;
  skiplist_item q = finger_search(v,key,l);
  if (l < 0) q = q->pred;
  return (q->height == MaxHeight) ? 0 : q;
 }

skiplist_item fskiplist::finger_locate_pred(GenPtr key) const
{ int l;
  skiplist_item q = finger_search(key,l);
  if (l < 0) q = q->pred;
  return (q==header) ? 0 : q;
 }

skiplist_item fskiplist::finger_lookup(skiplist_item v, GenPtr key) const
{ int l;
  skiplist_item q = finger_search(v,key,l);
  return (l<0) ? 0 : q;
 }

skiplist_item fskiplist::finger_lookup(GenPtr key) const
{ int l;
  skiplist_item q = finger_search(key,l);
  return (l<0) ? 0 : q;
 }
```

6 Insertions and Deletions

We discuss the various procedures to insert into or to delete from a skip list. The procedure *insert_item_at_item*(q, p, dir) inserts the item q after and before p respectively as prescribed by dir. The *true_height* of the header is also adjusted to the maximum of the old height and 1 plus the height of the new item.

The running time of *insert_item_at_item* is proportional to the height of the new item. The expected height of the new item is constant.

⟨*insert ...*⟩≡

```
void fskiplist::insert_item_at_item(skiplist_item q, skiplist_item p, int dir)
{  int k = q -> height;

  if (dir == before) p = p->pred;

  /* insert item |q| immediately after item |p| */
```

```
  register skiplist_item x;
  q->pred = p;
  p->forward[0]->pred = q;
  for (k = 0; k<=q->height; k++ )
  { while (k > p->height) p = p->backward;
    x = p->forward[k];
    if (p-> height == MaxHeight && x -> height < 0)
    {/* we have reached |header| and |STOP| and need to
           increase |true_height| */
     ((large_item) p) -> true_height = k + 1;
     p->forward[k+1 ] = x;
    }
    q->forward[k] = x;
    p->forward[k] = q;
    if (x->height == k) x->backward = q;
   }
   q->backward = p;

}
```

insert_at_item(p, key, inf) modifies the skip list in the vicinity of item p. If p's key is equal to *key* then its information is changed to *inf*. Otherwise a new item is created and inserted before or after p as dictated by *key*. The height of the new node is chosen randomly by a call *randomLevel*( ). The expected running time is constant.

$\langle$*insert ...*$\rangle$+≡

```
skiplist_item fskiplist::insert_at_item(skiplist_item p, GenPtr key, GenPtr inf)
{ register skiplist_item q;
  if (p->height  < 0) p = p->pred;
  else { if (p->height  < MaxHeight )
         { int c = cmp(key,p->key);
           if (c == 0)
              { clear_inf(p->inf);
                copy_inf(inf);
                p->inf = inf;
                return p;
               }
           if (c<0) p = p->pred;
         }
       }
  int k = randomLevel();
  if (k >= MaxHeight)
    { error_handler(0,"Warning: random level not below MaxHeight.\
 Level truncated to MaxHeight - 1");
      k = MaxHeight -1;
    }
  q = new fskiplist_node;
  q -> forward = new skiplist_item[k+1];
  q -> height = k;
  copy_key(key);
  copy_inf(inf);
  q->key = key;
  q->inf = inf;
```

```
  insert_item_at_item(q,p,after);

  return q;
}

int fskiplist::randomLevel()
{  register int height = 0;
   register int b  = 0;

   //if (prob==0.25)
      /* the following loop is equivalent to
         while (rand_int() < MAXINT/4) height++; */
      while (b==0)
      { b = randomBits&3;    // read next two random bits
        randomBits >>= 2;
        randomsLeft -= 2;
        if (b==0) height++;   // increase height with prob 0.25
        if (randomsLeft < 2)
        { randomBits = rand_int(0,MAXINT-1);
          randomsLeft = BitsInRandom;
         }
       }
/*
   else            // user defined prob.
     while (rrandom() < prob) height++;
*/

   return height;
 }
```

There is also a version of *insert_at_item* which inserts before or after p as directed by *dir*. The expected running time is again constant.

$\langle$*insert ...*$\rangle$+$\equiv$

```
skiplist_item fskiplist::insert_at_item(skiplist_item p, GenPtr key, GenPtr inf,
int dir)
{ register skiplist_item q;
  register int k = randomLevel();
   q = new fskiplist_node;
   q -> forward = new skiplist_item[k+1];
   q -> height = k;
   copy_key(key);
   copy_inf(inf);
   q->key = key;
   q->inf = inf;

   insert_item_at_item(q,p,dir);

   return q;
 }
```

remove_item removes an item and *del_item* removes an item, frees its storage and also adjusts the height of the skip list if required. The first function is used in the second and in *reverse_items*. A call *reverse_items*(p, q) with p equal or left of q reverses the subsequence with endpoints p and q.

insert(k, i) inserts a new item $\langle k, i\rangle$ or changes the information of the item with key k (if there is such an item) and *del*(k) removes the item with key k.

reverse_item has expected running time $O(d)$, where d is the length of the subsequence to be reversed. The other functions run in constant expected time.

⟨*insert ...*⟩+≡

```
void fskiplist::remove_item(skiplist_item q)
{ if (q->height == MaxHeight || q->height < 0)
   error_handler(1,"cannot remove improper item");

  register int k;
  register skiplist_item p = q->backward;
  register skiplist_item x;

  for(k=q->height; k>=0; k--)
  { while (p->forward[k] != q) p = p->forward[k];
    x = q->forward[k];
    p->forward[k] = x;
    if (x->height==k) x->backward = p;
   }
  x->pred = p;
}

void fskiplist::del_item(skiplist_item q)
{ if (q->height == MaxHeight || q->height < 0)
       error_handler(1,"cannot delete improper item");
  remove_item(q);
  clear_key(q->key);
  clear_inf(q->inf);
  skiplist_item p = q->forward[q->height];
  delete q->forward;
  delete q;
  if (p->height < 0)
   { large_item r = (large_item) p->backward;
    int& h = r -> true_height;
    while( h > 0 && r->forward[h - 1] == p) h--;
   }
}

skiplist_item fskiplist::insert(GenPtr key, GenPtr inf)
{ int k;
  skiplist_item p = search(header,header->true_height,key,k);
  if (k >= 0)
    {clear_inf(p->inf);
     copy_inf(inf);
     p->inf  = inf;
     return p;
    }
  p = insert_at_item(p,key,inf,before);
  return p;
 }

void fskiplist::del(GenPtr key)
{ int k;
  skiplist_item q = search(header,header->true_height,key,k);
  if (k>=0) del_item(q);
 }

void fskiplist::reverse_items(skiplist_item p, skiplist_item q)
{ skiplist_item r;
  while (p!=q)
  { r = p;
```

```
      p = p->forward[0];
      remove_item(r);
      insert_item_at_item(r,q,after);
    }
  }
```

7 Concatenate, Split, Merge and Delete_Subsequence

7.1 Concatenate

We describe how to concatenate two skiplists in time $O(\log \min(n_1, n_2))$ where n_1 and n_2 are the sizes of the two skiplists.

We first make sure that *this* skip list is the higher list (by swaping *header* and *STOP* of *this* list and *S1*, if necessary) and then append *S1* to either the front or the rear of *this* list.

Let's say we need to append *S1* to the rear of *this*. Imagine that *S1* is placed between the last element of *this* and *STOP* of *this* and then redirect pointers as required. More precisely, for any k less than the height of *S1* the k-th forward pointer out of the rightmost tower in *this* of height at least k is redirected to the first item in *S1* of height at least k and the k-th forward pointer out of the rightmost tower in *S1* of height at least k is redirected to *STOP* of *this*.

The running time of *conc* is proportional to the smaller of the two heights and is therefore $O(\log \min(n_1, n_2))$.

⟨*concatenate ...*⟩≡

```
void fskiplist::conc(fskiplist& S1, int dir)
{ if (header ->true_height < S1.header->true_height)
   { swap(header->myseq,S1.header->myseq);
     swap(header,S1.header);
     swap(STOP,S1.STOP);

     dir = ((dir == after) ? before : after);
   }

if (S1.STOP ->pred == S1.header)  return;

/* S1 is non-empty and since height >= S1.height. this is also non-empty */

  if (dir == after)
   { skiplist_item p = STOP ->pred;
     skiplist_item q = S1.STOP ->pred;

     if (cmp(p->key, S1.header->forward[0]->key) >=0)
        error_handler(1,"precondition of conc violated");

     STOP ->pred = q;
     S1.header -> forward[0] ->pred = p;

     for (int k = 0; k < S1.header->true_height; k++)
      { /* |p| and |q| are the rightmost items of height at least |k| in
           |this| and |S1| respectively */
        skiplist_item r = S1.header->forward[k];
        p->forward[k] = r;
        if (r->height == k) r->backward = p;
        q->forward[k] = STOP;
        while (p->height == k) p = p->backward;
        while (q->height == k) q = q->backward;
      }
    }
  else
```

```
  { skiplist_item q = S1.STOP ->pred;
    if (cmp(q->key, header->forward[0]->key) >=0)
       error_handler(1,"precondition of conc violated");
    S1.header->forward[0]->pred= (skiplist_item) header;
    header -> forward[0] ->pred = q;
    for (int k = 0; k < S1.header->true_height; k++)
      { /* |q| is the rightmost item of height at least |k|in |S1| */
       skiplist_item r = header->forward[k];
       q->forward[k] = r;
       if (r->height == k) r->backward = q;
       r = S1.header->forward[k];
       header->forward[k] = r;
       if (r->height == k) r->backward= (skiplist_item) header;
       while (q->height == k) q = q->backward;
      }
  }
 S1.header -> true_height = 0;
 S1.STOP -> pred = (skiplist_item) S1.header;
 S1.header-> forward[0] = S1.STOP;
#ifdef CHECK_INVARIANTS
this -> check_data_structure("this in conc");
/* in |delete_subsequence| we call |conc| with an argument |S1| that
does not
know the type of keys and infs. here it is important that the implicite
argument of the next instruction is |this|. */
check_data_structure(S1,"S1 in conc");
#endif
}
```

7.2 Split

S.split_at_item$(p, S1, S2, dir)$ splits the skip list, call it P, containing p before or after item p into lists *S1* and *S2* as directed by *dir* in time proportional to the logarithm of the shorter result. Clearly, *S1* and *S2* must be distinct, but one of them might be equal to P . If both of them are different from P then P is empty after the split. All four skiplists must have the same type. S may be any skip list; it only provides the type.

A method whose running time is proportional to the logarithm of the size of P is easy to describe. We simply erect two new improper towers before or after p.

In order to obtain the more efficient bound we have to reuse *header* and *STOP* of list P for the larger output. We first determine the lower of the two outputs by simultaneously walking from p and its successor (this assumes $dir \equiv after$) to *header* and *STOP* until one of the two walks reaches its destination. Let *max_level* be the maximal level reached, i.e., both sublists contain a tower of height *max_level* and for one of the sublists this is the maximal height.

Assume first *max_level* is the maximal height of a tower in *S1*. Then $1 + max\_level$ is the height of *S1* and *height* is the height of *S2* after the split. We interchange *header* and *STOP* of *S2* and P and then remove *S1* from *S2*. To remove *S1* from *S2* we do the following for each k, $0 \le k \le max\_level$. Let *p1* be the rightmost item in *S1* of height at least k. The k-th forward pointer out of *S1.header* is redirected to the destination of the k-th forward pointer out of *S2.header*, the k-th forward pointer out of *S2.header* is redirected to the destination of the k-th forward pointer out of *p1* and the k-th forward pointer out of *p1* is redirected to *S1.STOP*.

Assume next that *max_level* is the maximal height of a tower in *S2*. Then $1 + max\_level$ is the height of *S2* after the split and *height* is the height of *S1* after the split. We interchange *header* and

STOP of *S1* and *P* and then remove *S2* from *S1* in a way similar to the one described above.

The running time is $O(max\_level)$ and the expected value of *max_level* is $O(\log\min(n_1, n_2))$, where n_1 and n_2 are the sizes of the two parts respectively.

$\langle$*concatenate ...*$\rangle{+}{\equiv}$

```
void fskiplist::split_at_item(skiplist_item p,fskiplist& S1 ,fskiplist& S2,int dir)
{ if (dir == before) p = p->pred;

  skiplist_item p1 = p;
  skiplist_item p2 = p ->forward[0];
  int max_level = -1;

  while ( p1->height  < MaxHeight && p2->height  >= 0)
   { /* |p1| and |p2| are proper towers of height larger than |max_level| */
     max_level++;
     while (p1 -> height == max_level)  p1 = p1 -> backward;
     while (p2 -> height == max_level)  p2 = p2 -> forward[max_level];
   }

  /* we have seen proper towers of height |max_level| on both sides
     of the split and either |p1| or |p2| is a sentinel */

  large_item pheader;

  if (p1->height == MaxHeight)
      pheader = (large_item) p1;
  else
      pheader = (large_item) p2->backward;

  fskiplist* Pp = pheader ->myseq;

  if (Pp != &S1)  S1.clear();
  if (Pp != &S2)  S2.clear();

  if (p1->height == MaxHeight)
    { /* we reuse |pheader| and |pSTOP| for |S2| */
     if (Pp != &S2)
      { swap(Pp->header->myseq, S2.header -> myseq);
        swap(Pp->header,S2.header);
        swap(Pp->STOP,S2.STOP);
      }
     S1.header -> true_height = 1+max_level;

     p1 = p;

    for (int k =0; k <= max_level; k++)
      { /* |p1| is the rightmost item in |S1| of height at least |k| */
        skiplist_item q = S2.header->forward[k];
        S1.header->forward[k] = q;
        if (q -> height == k) q->backward = (skiplist_item) S1.header;
        S2.header->forward[k] = p1 -> forward[k];
        if (p1 -> forward[k] -> height == k) p1->forward[k] -> backward =
                     (skiplist_item) S2.header;
        p1->forward[k] = S1.STOP;
        while (k == p1->height) p1 = p1 -> backward;
       }

     S1.header->forward[max_level + 1] = S1.STOP;
     /* the next line sets the predecessor of |S1.STOP| correctly if |S1| is
        non-empty; if it is empty the last line corrects the mistake */
      S1.STOP ->pred = p;
      S2.header->forward[0] ->pred = (skiplist_item) S2.header;
```

```
        S1.header->forward[0] ->pred = (skiplist_item) S1.header;
      }
    else
      { /* we want to reuse |pheader| and |pSTOP| for |S1| */
        if (Pp != &S1)
         { swap(Pp->header->myseq,S1.header->myseq);
           swap(Pp->header,S1.header);
           swap(Pp->STOP,S1.STOP);
         }
        S2.header -> true_height = 1 + max_level;

        p1 = p;
        p2 = S1.STOP ->pred;

      for (int k =0; k <= max_level; k++)
        { /* |p1| and |p2| are the rightmost items in |S1| and |S2|
             of height at least |k| respectively */

          skiplist_item q = p1->forward[k];
          S2.header->forward[k] = q;
          if (q -> height == k) q->backward = (skiplist_item) S2.header;
          p1->forward[k] = S1.STOP;
          p2->forward[k] = S2.STOP;
          while (k == p1->height) p1 = p1 -> backward;
          while (k == p2->height) p2 = p2 -> backward;
         }

      S2.header->forward[max_level + 1] = S2.STOP;
      /* the next line sets the predecessor of |S2.STOP| correctly if |S2|
         is non-empty; if it is empty then the next
         line corrects the mistake */
      S2.STOP ->pred = S1.STOP->pred;
      S2.header->forward[0] ->pred = (skiplist_item) S2.header;
      S1.STOP ->pred = p;
      S1.header->forward[0] ->pred = (skiplist_item) S1.header;
     }

   if (Pp != &S1 && Pp != &S2)
    { /* |P| is empty if distinct from |S1| and |S2| */
      Pp->header->forward[0] = Pp->STOP;
      Pp->STOP ->pred = Pp->STOP -> backward= (skiplist_item) Pp->header;
      Pp->header->true_height = 0;
    }
#ifdef CHECK_INVARIANTS
this -> check_data_structure("this in split");
Pp -> check_data_structure("P in split");
check_data_structure(S1,"S1 in split");
check_data_structure(S2,"S2 in split");
#endif
}
```

7.3 Merge

We desribe how to merge two skip lists of length $n1$ and $n2$ in time $O(\binom{n1+n2}{n1})$.

We first determine the shorter list (by stepping through both lists in lock-step fashion) and make sure that *this* list is the larger list (by interchanging *header* and *STOP* of *this* list and *S1* otherwise). We then erect a finger at the first item of *this* list and consider the items of *S1* one by one. We locate the item by a finger search, insert the item into *this* list and advance the finger to the point of insertion.

For the running time analysis we assume without loss of generality that $n_1 \le n_2$. For i, $1 \le i \le n_1$, let d_i be the stride of the finger search when inserting the i-th item of *S1* into S. Then $n_2 = \sum_i d_i$ and the total running time is $O(n_1 + \sum_i \log d_i)$. This sum is maximal if all the d_i's are equal to n_2/n_1 and is hence bounded by $O(n_1(1+\log(n_2/n_1))) = O(\binom{n1+n2}{n1})$.

$\langle$*concatenate ...*$\rangle{+}\equiv$

```
void fskiplist::merge(fskiplist& S1)
{ skiplist_item p= (skiplist_item) header;
  skiplist_item q = S1.header;
  while (p->height  >= 0 && q ->height >= 0)
   { p = p->forward[0];
     q = q->forward[0];
   }

  if (q ->height >= 0)
   { /* swap if |this| is shorter than |S1| */
     swap(header->myseq,S1.header->myseq);
     swap(header,S1.header);
     swap(STOP,S1.STOP);
   }

  /* now |S1| is at most as long as |this| */
  skiplist_item finger= (skiplist_item) header;
  p = S1.header -> forward[0];

  while (p ->height >= 0)
    { skiplist_item q = p -> forward[0];
      int l;
      finger = finger_search(finger,p->key,l);
      if (l >= 0) error_handler(1,"equal keys in merge");
      insert_item_at_item(p,finger,before);
      finger = p; // put finger at newly inserted item
      p =q;
    }

  S1.header->true_height = 0;
  S1.STOP -> pred = (skiplist_item) S1.header;
  S1.header-> forward[0] = S1.STOP;
#ifdef CHECK_INVARIANTS
check_data_structure("this in merge");
S1.check_data_structure("S1 in merge");
#endif

}
```

7.4 Delete Subssequence

We desribe how to delete a subsequence from a skip list. More precisely, if a and b are items in a list P with a left of or equal to b then the call $S.delete\_subsequence(a, b, S1)$ deletes the subsequence starting at a and ending at b from P and assigns it to *S1*. The running time is $O(\log \min(n_1, n - n_1))$ where n and n_1 are the length of P and *S1* respectively. S only provides the type.

The items a and b split P into three parts. We first determine the lowest of the parts by simultaneously walking from $a \rightarrow pred$ and b to the left and from $b \rightarrow forward[0]$ to the right until we reach *header*, a tower left of a, or *STOP* respectively.

If either the first or the last subsequence is lowest then the operation can be reduced to two splits and one conc. If what is to become *S1* is lowest we directly insert *S1*'s *header* and *STOP* before a and after b respectively.

Let h_i be the height of the i-th part. Then $E[h_2] = \log n_1$ and $\max(E[h_1], E[h_3]) = \log n - n_1$. The time to determine the lowest part is $\min(h_1, h_2, h_3)$. If h_2 is smallest then the running time of actually deleting the subsequence is $O(h_2)$. If h_2 is not the smallest then the times for the two splits and one conc are $\min(\max(h_1, h_2), h_3))$, $\min(h_1, h_2)$, and $\min(h_1, h_3)$ respectively. All three quantities are bounded by $\min(h_2, \max(h_1, h_3))$. The expected running time is therefore $O(\log \min(n_1, n - n_1))$ in both cases.

$\langle$*concatenate ...*$\rangle$+≡

```
void fskiplist::delete_subsequence(skiplist_item a,skiplist_item b,fskiplist& S1)
{ S1.clear();
  skiplist_item p1 = a -> pred;
  skiplist_item p2 = b;
  skiplist_item p3 = b ->forward[0];
  int k = -1;

  while ( p1->height  < MaxHeight && p3->height  >= 0 &&
       p2->height  < MaxHeight && cmp(p2 -> key, a->key) >= 0)
   { k++;
     while (p1 -> height == k)  p1 = p1 -> backward;

     while ( p2 ->height == k)  p2 = p2 -> backward;

     while ( p3 -> height == k)  p3 = p3 -> forward[k];
   }

  if (p1->height == MaxHeight || p3->height  < 0)
   { if (p1 -> height < MaxHeight) p1 = p3 -> backward;
     fskiplist* Pp = ((large_item) p1) -> myseq;
     fskiplist S2,S3;

     split_at_item(b,S2,S3,after);

     /* The next line is subtle. Note that |a| is an element of |S2|
        and that we split |S2| into |Pp| and |S1|. So why do we use
        |this| as the implicite argument for the split instead of |S2|.
        The reason is that |S2| is an ordinary skip list and has no
        knowledge of the key and the inf type. In particular |S2|'s
        virtual functions (e.g. |cmp|) are as defined in fskiplist.h.
        Thus |S2| cannot properly compare two keys. It does not need to
        in |split_at_item|. However, during program testing |split_at_item|
        may want to call |check_data_structure| and this procedure needs
        to compare keys. So it is essential that we use |this| as the
        implicite argument. */

     split_at_item(a,*Pp,S1,before);

     Pp->conc(S3,after);

     return;
   }

  /* the middle list is the lowest and we have to do some work */

  p1 = a->pred;
  p2 = b;

  /* correct predecessor pointers */

  a->pred = (skiplist_item) S1.header;
  S1.STOP -> pred = b;
```

```
    b->forward[0]->pred = p1;

    /* height of |S1| */

    S1.header->true_height = 1 + k;
    S1.header->forward[1+k] = S1.STOP;

    for (int i = 0; i <= k; i++)
      { /* |p1| and |p2| are the rightmost items of height at least
        |i| in the first and the second part of the list respectively */

        skiplist_item q = p1 ->forward[i];
        S1.header->forward[i] = q;
        if (q -> height == i) q->backward = S1.header;
        q = p2 ->forward[i];
        p1 ->forward[i] = q;
        if (q -> height == i) q->backward = p1;
        p2 ->forward[i] = S1.STOP;

        while (i == p1->height)  p1 = p1 -> backward;
        while (i == p2->height)  p2 = p2 -> backward;
      }
  }
```

8 Miscellaneous and Inline Functions

We close with some miscellaneous functions. *size* returns the number of elements in a skiplist. It runs in linear time.

check_data_structure checks whether a data structure is legal skip list. There are two versions of this procedure. The version *S.check_data_structure*$(S1, s)$ checks $S1$ and in case of error prints the control string s; the argument S is only used for the type information. This version is only for internal use and is appropriate for locally defined skiplists (as they have no knowledge of the type of the underlying sorted sequence), cf. the implementation of *delete_subsequence*. The version *S.check_data_structure*(s) is equivalent to *S.check_data_structure*$(S1, s)$ and is directly defined in the header file. *check_data_structure* runs in time $O(n \log n)$.

print prints the elements of a skiplist. Again there are two versions. The call *S.print*$(S1, o, s, c)$ prints the elements of $S1$ onto the ostream o using c as the separating character and using s as the introductory text; S provides the type information. The other version is directly defined in the header file.

⟨*miscellaneous*⟩≡

```
int fskiplist::size()
{ int count = 0;
  skiplist_item p;
  p= (skiplist_item) header;
  while ((p = p->forward[0]) != STOP)  count++ ;
  return count;
}

skiplist_item fskiplist::pred(skiplist_item p) const
{ skiplist_item q =  p->pred;
  return (q->height == MaxHeight) ? 0 : q;
}

void fskiplist::check_data_structure(const fskiplist& S1,string s)
{ string t = "";

  if (S1.header != S1.header -> myseq -> header)
```

```
      {t =t + "error in myseq";}
    int h = -1;
    skiplist_item p= (skiplist_item) S1.header;
    while (p != S1.STOP)
     { skiplist_item q = p->forward[0];
      if (p != q ->pred)  t =t + "error in underlying list";
      if (p != S1.header && p->height > h) h = p->height;
      if (p != S1.header && q != S1.STOP && cmp(p->key,q->key)>=0)
             { t = t + "error in order";
       print(S1,cout,"I detected an error in the order. Here is the list",' ');
               newline;
             }
      if (p != S1.STOP)
        { for (int i = 0; i <= Min(p->height,S1.header->true_height); i++)
           { skiplist_item r = p ->forward[0];
             while (r ->height < i && r != S1.STOP) r = r ->forward[0];
             if ( r != p->forward[i])  t =t + "error in forward";
             if ( i == r -> height && r -> backward != p)
                          t = t + "error in backward";
           }
        }
      p = q;
     }
    if (S1.header ->true_height < h +1)  t = t + "height too small";
    if (S1.header -> true_height > h +1)  t =t+  "height too large";
    if (S1.STOP -> backward != (skiplist_item) S1.header)  t =t + "error in stop";
    string t1 = "";
    if (t != t1) {cout << s << "   " << t; newline; cout.flush();}
  }

  void fskiplist::print(const fskiplist & S1,ostream& out,string s, char space) const
  { skiplist_item p= ((skiplist_item) S1.header) -> forward[0];
    out << s << "\n";
    while (p != S1.STOP)
    {  print_key(p->key);
       out << string(space);
       p = p ->forward[0];
    }
    out.flush();
  }
```

We conclude with the list of inline functions. They are self-explaining.

⟨*implementation of inline ...*⟩≡

```
  inline GenPtr  fskiplist::key(skiplist_item p)  const  {  return p->key;  }
  inline GenPtr  fskiplist::inf(skiplist_item p)  const  {  return p->inf;  }
  inline GenPtr& fskiplist::info(skiplist_item p) const  {  return p->inf;  }
  inline int     fskiplist::get_height(skiplist_item p) const  {  return p->height;  }

  inline skiplist_item fskiplist::first_item() const
  { skiplist_item q = header->forward[0];
    return (q==STOP) ? 0 : q;
```

```
 }
inline skiplist_item fskiplist::min_item() const
{ skiplist_item q = header->forward[0];
  return (q==STOP) ? 0 : q;
 }
inline skiplist_item fskiplist::max_item() const
{ skiplist_item q = STOP->pred;
  return (q==header) ? 0 : q;
 }
inline skiplist_item fskiplist::next_item(skiplist_item p) const
{ skiplist_item q =  p->forward[0];
  return (q->height < 0) ? 0 : q;
 }
inline skiplist_item fskiplist::succ(skiplist_item p) const
{ skiplist_item q =  p->forward[0];
  return (q->height < 0) ? 0 : q;
 }

inline void fskiplist::change_inf(skiplist_item p, GenPtr inf)  {  p->inf = inf;  }
inline int  fskiplist::empty() const  {  return (header->forward[0] == STOP);  }
inline skiplist_item fskiplist::find_min() const  {  return min_item();  }
inline void fskiplist::del_min()
{ skiplist_item p = min_item(); if (p) del_item(p); }
inline void fskiplist::decrease_key(skiplist_item p, GenPtr k)
{insert(k,p->inf); del_item(p);}

/* dummy I/O and cmp functions */
inline void Print(const fskiplist&,ostream&)  {  }
inline void Read(fskiplist&, istream&)  {  }
inline int  compare(const fskiplist&,const fskiplist&)  {  return 0;  }
```

9 Test Routines

We define two test programs. The first program checks all operations on sorted sequences with integer keys and informations. The second program uses a non-trivial type for the keys. In both cases we use comments in the program to give more detail.

⟨*sortseq-test.c*⟩≡

```
#include <LEDA/basic.h>
#include "sortseq.h"
#include </KM/usr/mehlhorn/Programs/basic.h>
main()
{sortseq<int,int> S;
sortseq<int,int> *A[1000];
int i;
report("decls worked");
for (i = 0; i <1000; i++)
 {int x = rand_int(0,10000);
 while (S.lookup(x)!=nil)  x = rand_int(0,10000);
 S.insert(x,0);
```

```
 if (S.lookup(x) == nil || compare( S.key(S.lookup(x)),x) != 0 )
      error_handler(1, "insert or lookup");
 A[i] = new sortseq<int,int>;
 A[i]->insert(x,0);
 }

S.check_data_structure("after inserts");

report("insertion worked");

/* we have now a sorted sequence |S| of 1000 items and an array |A| of
1000 sorted sequences of one item each. Merging the 1000 sequences into
a single sequence should produce a copy of |S|.\\
\smallskip
We merge the 1000 sequences in random order.
*/

for (i = 999; i >= 1; i--)
{int j = rand_int(0,i);
swap(A[j],A[i]);
j = rand_int(0,i-1);
swap(A[j],A[i-1]);
A[i-1] -> merge(* A[i]);
}

report("merge ran to completion");

/* we compare |S| and |A[0]| element by element */

if (S != (*(A[0]))) error_handler(1,"error in merge");

report("merge correct");

/* we next test the various search routines. */

seq_item finger;

for( i =0; i <10000; i++)
 {int x = rand_int(0,10000);
  finger = S.locate_pred(x);
  if (finger == nil) finger = S.min_item();
  x = rand_int(0,10000);
  if (S.lookup(x) != S.finger_lookup(finger,x))
       error_handler(1,"error in lookup");
  if (S.locate_pred(x) != S.finger_locate_pred(finger,x))
       error_handler(1,"error in locate_pred");
  if (S.locate_succ(x) != S.finger_locate_succ(finger,x))
       error_handler(1,"error in locate_succ");
  if (S.lookup(x) != S.finger_lookup(x))
            error_handler(1,"error in lookup: dove-tailed");
  if (S.locate_pred(x) != S.finger_locate_pred(x))
       error_handler(1,"error in locate_pred: dove-tailed");
  if (S.locate_succ(x) != S.finger_locate_succ(x))
       error_handler(1,"error in locate_succ: dove-tailed");

  }

report("lookups worked");

/* next comes |split| and |conc|. We split |S| at a random location and then
concatenate the two pieces again. We do so 100 times. Finally, we check the
result against |A[0]| */
```

```
for (i =0; i <100; i++)
{int x = rand_int(0,10000);
finger = S.locate_pred(x);
if (finger == nil) finger = S.min_item();

sortseq<int,int> S1;
sortseq<int,int> S2;
S.split(finger,S1,S2,after);

S1.conc(S2,after);
S.conc(S1,after);
}

if (S != (*(A[0]))) error_handler(1,"error in conc or split");

report("split and conc worked");

/* next comes |delete_subsequence|. We create two fingers and delete the
subsequence between the two fingers. Then we use |split| and |conc| to
rebuild the original sequence. We check by comparing to |A[0]|. */

finger = S.min_item();
for (i= 0; i <= 100; i++) finger = S.succ(finger);
seq_item finger2 = finger;
for (i= 0; i <= 100; i++) finger2 = S.succ(finger2);

seq_item finger_pred = S.pred(finger);

sortseq<int,int> S1;
sortseq<int,int> S2;
sortseq<int,int> S3;

S.delete_subsequence(finger,finger2,S2);
S.split_at_item(finger_pred,S1,S3,after);
S.conc(S1,after);
S.conc(S2,after);
S.conc(S3,after);

if (S != (*(A[0]))) error_handler(1,"error in delete subsequence");

}
```

The second test program checks whether keys are handled correctly. We define data type *pair* (a pair of doubles) and *Pair* (a pointer to a *pair*) and use both of them as keys.

$\langle$*sortseq-test2.c*$\rangle\equiv$

```
#include <LEDA/basic.h>
#include "sortseq.h"
#include </KM/usr/mehlhorn/Programs/basic.h>

class pair
{double x,y;

public:
pair() {  x = y =0; }
pair(const pair& p) { x =p.x;  y =p.y; }
pair(double a,double b) { x =a; y=b; }
friend void Print(const pair& p,ostream& o) { o << p.x << "  " << p.y <<"\n"; }
friend void Read(pair& p, istream& i) { i >> p.x; i >> p.y; }
friend int compare(const pair& p,const pair& q)
  {if (p.x < q.x) return -1;
    else if (p.x == q.x) return 0;
```

```
        else return +1;
  }
};

typedef pair* Pair;

int compare(const Pair& p,const Pair& q)
 { return compare(*p,*q); }

void Print(const Pair& p,ostream& o) { Print(*p,o); }

main()
{
report("first test");
{sortseq<pair,int> S;
pair p1(4,0);
pair p2(2,0);
pair p3(6,0);
seq_item sit = S.insert(p1,0);
seq_item sit1 =S.insert_at(sit,p2,0,before);
if (compare(S.key(sit1), p2)!=0) report("error 1");
sit1 = S.insert_at(sit,p3,0,after);
if (compare(S.key(sit1), p3) != 0) report("error 2");
S.check_data_structure("pair");
S.print("here comes S"); newline;

sortseq<pair,int> S1;
S.delete_subsequence(sit1,sit1,S1);
S.print("S"); newline;
S1.print("S1"); newline;

/* copy constructor */
sortseq<pair,int> S2(S);

if (S != S2)  error_handler(1,"pair: error in copy constructor");
S2.check_data_structure("after copy constructor");

/* assignment */
S2 = S;
if (S2 != S)  error_handler(1,"pair: error in assignment");
S2.check_data_structure("after assignment");

}
report("second test");

{sortseq<Pair,int> S;
Pair p1 = new pair(4,0);
Pair p2 = new pair(2,0);
Pair p3 = new pair(6,0);
seq_item sit = S.insert(p1,0);
seq_item sit1 =S.insert_at(sit,p2,0,before);
if (compare(S.key(sit1), p2)!=0) report("error 1");
sit1 = S.insert_at(sit,p3,0,after);
if (compare(S.key(sit1), p3) != 0) report("error 2");
S.check_data_structure("Pair");
S.print("here comes S"); newline;

sortseq<Pair,int> S1;
S.delete_subsequence(sit1,sit1,S1);
```

```
S.print("S"); newline;
S1.print("S1"); newline;
}
}
```

10 Implementation History

Skip lists were invented by W. Pugh[Pug89, Pug90]. Stefan Näher implemented them for LEDA in 1991. Kurt Mehlhorn added finger searches and operations *conc*, *split*, *merge*, and *delete_subsequence* in August and September 94.

References

[MN94] K. Mehlhorn and St. Näher. *LEDA - A Platform for Combinatorial and Geometric Computing*. 1994. in preparation.

[N9̈5] St. Näher. LEDA Manual Version 3.1. Technical Report MPI-I-95-1-002, Max-Planck-Institut für Informatik, 1995.

[Pug89] W. Pugh. A skip list cookbook. Technical Report 89-72.1, University of Maryland, Department of Computer Science, Insitute for Advanced Computer Studies, 1989.

[Pug90] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.